Numpy

Numpy, which stands for numerical Python, is a Python library package to support numerical computations. The basic data structure in numpy is a multidimensional array object called ndarray. Numpy provides a suite of functions that can efficiently manipulate elements of the ndarray.

1. Importing numpy and setting alias for np
    - import numpy as np

1 dimensional Array

In computer science, an array data structure, or simply an array, is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.
One dimensional array is also known as linear array. One dimensional array consist of single list of similar data

1. Creating an numpy array
    - array_01 = np.array([1, 2, 3,8])
2. Know about the dimension of that array
    - np.ndim(array_01)
3. Know the shape of the array. np.shape is used to see the structure of that array i.e. how many rows or columns it has
    - np.shape(array_01)

2 Dimensional Array

2d array represent multiple data items as table that consist of rows and columns

1. Create multiple dimensional array
    - array_02 = np.array([[1, 2, 3], [4, 5, 6]])

2. Know about the dimension of that array
    - np.ndim(array_02)
3. Know the shape of the array.
    - np.shape(array_02)

Mathematical Operation on Array

Matrix multiplication of the array can be performed with two methods one using '@' operator and another with the numpy function

1) First let's make two 2dimensional array
    - np_01 = np.array([[1, 2], [4, 5]])
    - np_02 = np.array([[7, 8], [10, 11]])
2) Printing any array we will get the following result

```
1 np_01
```

```
array([[1, 2],
       [4, 5]])
```

3) Now lets look at the shape, dimension and type of that array

    - print('Arrays shape --> {}, dimension --> {} , Type --> {}'.format(np.shape(np_01),np.ndim(np_01),type(np_01)))

      The result should look something like this
      "Arrays shape --> (2, 2), dimension --> 2 , Type --> <class 'numpy.ndarray'>"
4) Now lets perform the array multiplication with **@** sign
    - multiply_array = np_01 @ np_02

      The result should look like this

```
array([[27, 30],
       [78, 87]])
```

5) Lets perform the same multiplication with **np.dot** function of numpy
    - multiply_array_02 = np.dot(np_01, np_02)

      Result

```
array([[27, 30],
       [78, 87]])
```

Simple Multiplication

This multiplication is not matrix multiplication but a simple element wise multiplication

1. Multiplication using * sign
   - multiply_array_03 = np_01 * np_02
   - Result

```
array([[ 7, 16],
       [40, 55]])
```

2. Multiplication using np.multiply function
   - multiply_array_04 = np.multiply(np_01, np_02)

```
array([[ 7, 16],
       [40, 55]])
```

Note: Matrix multiplication and simple multiplication are not the same. Evaluate the difference with the result.

Other arithmetic operations

1. Adding with + sign
   - sum_array_01 = np_01 + np_02

```
array([[ 8, 10],
       [14, 16]])
```

2. Adding with np.add ()
   - sum_array_02 = np.add(np_01, np_02)

```
array([[ 8, 10],
       [14, 16]])
```

3. Subtracting array
   - substract_array_01 = np_01 - np_02

```
array([[-6, -6],
       [-6, -6]])
```

4. Getting all the sum of matrix
   - element_sum = np.sum(np_01)
     Result = 12

Broadcasting

The term broadcasting refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements. If two arrays are of exactly the same shape, then these operations are smoothly performed.

1. Add certain number to each element of the array
   - nump_broad = np_01 + 3

```
array([[4, 5],
       [7, 8]])
```

2. Add an array to the existing array
   - nump_broad_02 = np_01 + np.array([7, 8])

```
array([[ 8, 10],
       [11, 13]])
```

3. Divide the array by 2 using np.divide function
   - divide_01 = np.divide([10, 20, 30], 2)

```
3 divide_01
```

```
array([ 5., 10., 15.])
```

   - print(divide_01.dtype) | Check the type of that array

```
1 print(divide_01.dtype)
```

```
float64
```

4. Floor Divide: floor_divide is division except that it returns the largest possible int value. Result of the integer value is always integer
   - divide_02 = np.floor_divide([10, 20, 30], 2)

```
1 print(divide_02.dtype)
```

```
int32
```

5. Divide matrix by matrix
   - divide_03 = np.divide(np_01, np_02)

```
1 divide_03
```

```
array([[0.14285714, 0.25      ],
       [0.4       , 0.45454545]])
```

Random function

Random functions are very useful for generating random numbers. We can give specific range as well for generating random numbers

1. Generating a random ***floating*** number
   - np.random.rand()

   Result: The result can be any number every time the function is executed. Its type will be float number
2. Generating a random integer number with some range
   - random_interger_array = np.random.randint(1, 50, (3,4))

```
array([[35, 27, 32, 22],
       [19, 15, 36, 10],
       [42, 12, 39, 31]])
```

   This matrix will also vary in every execution but values will in range of 1 to 50

Zeros and ones

There may be some cases where we might one to create a matrix with a certain shape with zeros or ones. Let's look how to create them

1. Create an array of a certain size having all elements 0.
   - zeros = np.zeros((3,4))

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

2. Creating an array consisting of 1
   - ones = np.ones((3,4))

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Some filtering

1. Filtering array with np.logical_and() function
   - filter_array = np.logical_and(random_interger_array > 30, random_interger_array < 50)

```
1 filter_array
```

```
array([[ True, False,  True, False],
       [False, False,  True, False],
       [ True, False,  True,  True]])
```

2. Getting all the values that are true from previous filter

- filter_random_Arr = random_interger_array[filter_array]

```
array([35, 32, 36, 42, 39, 31])
```

Statistics

Let's look at some statistical operation numpy allows us to perform

1. Create an array
   - my_data = np.array([1, 3, 4, 5, 6, 7, 8])
2. Find the mean of that array
   - mean = np.mean(my_data)

```
3 mean
```

```
: 4.857142857142857
```

3. Find the median of that array
   - median = np.median(my_data)

```
3 median
```

```
5.0
```

4. Getting the variance of that data
   - var = np.var(my_data)

```
1 var
```

```
4.97959183673469
```

5. Calculating the standard deviation of that variance
   - sd = np.math.sqrt(var)

```
3 sd
```

```
2.2314999074019015
```

Distribution

In this section we will look at different way of creating distribution with numpy

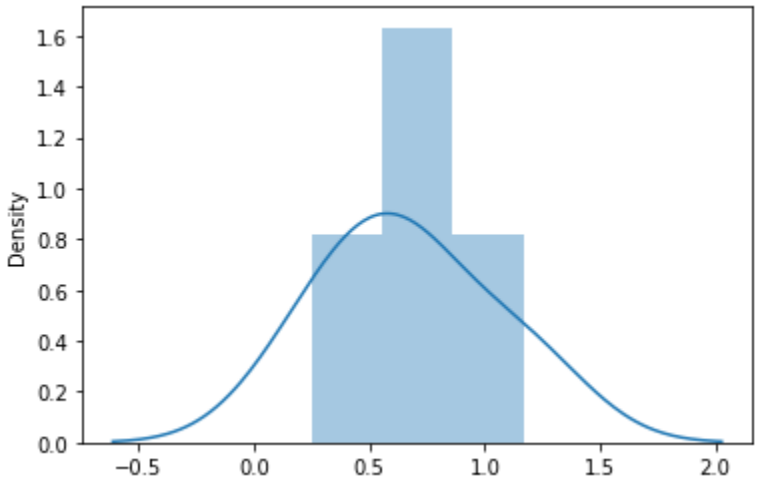
Np.random : np.random is used to generate random value for certain function

1. Creating a normal distribution value for 4 numbers
   - normal_distribution = np.random.standard_normal(4)

```
1 normal_distribution
```

```
array([1.16635403, 0.66283585, 0.5944263 , 0.24790702])
```

2. To visualize that data lets plot it using seaborn. We will further learn about seaborn as well so don't panic for now :)
   - import seaborn as sns
   - sns.distplot(normal_distribution)



3. Creating a uniform distribution
   - uniform_distribution = np.random.uniform(1, 50, (3,4))

     Here first 1 and 50 value are the range upto wich we want to generate random variables and (3,4) is the matrix shape we want to generate

```
1 uniform_distribution
```

```
array([[34.74032363,  7.5708202 , 39.20114313, 40.72827908],
       [13.84978316, 10.47046965, 22.97040373,  5.24943243],
       [25.70642009,  1.08604862, 14.42508345, 46.30948555]])
```

   - Visualizing the values

: <AxesSubplot:ylabel='Density'>